Lab 6 Report

1st Skyler Cook
Electrical and Computer Engineering
Utah State University
Logan, USA
A02304185@aggies.usu.edu

2nd Jaron Seedal
Electrical and Computer Engineering
Utah State University
Logan, USA
a02358703@aggies.usu.edu

3rd Dr. Jonathan Philips
Electrical and Computer Engineering
Utah State University
Logan, USA
jonathan.phillips@usu.edu

***Abstract*—This project implements a dual-clock accumulator system on the DE10-Lite FPGA using VHDL. The design expands on the previous 24-bit accumulator by introducing two finite-state machines (FSMs), a clock-domain-crossing FIFO, and a PLL. The 5 MHz FSM handles button inputs and writes switch values into the FIFO, while the 12.5 MHz FSM reads from the FIFO and performs accumulation. When five values are stored, they are added to the running 24-bit total displayed in hexadecimal on the six 7-segment displays.**

I. Introduction

This lab builds upon the accumulator design from Lab 5 by adding multiple clock domains and inter-FSM communication through a FIFO. The project demonstrates how to manage timing and data transfer between two systems running on different clocks using a PLL-generated 5 MHz and 12.5 MHz clock. One push button serves as an active-low reset, and the other acts as a store control to push the 10-bit switch value into the FIFO. After five entries are stored, the FIFO automatically drains, and the values are added to the 24-bit accumulator. The accumulated total is displayed in hexadecimal across the six 7-segment displays, while the LEDs mirror the switch positions. This implementation highlights the principles of clock-domain synchronization, FSM coordination, and reliable data buffering in FPGA design.

II. Objectives

The objectives of the lab were:

1. One push button is the "reset" button. Pressing this button clears the accumulated value.
2. The 24-bit accumulated value is displayed in hexadecimal on the six 7-segment displays.
3. The other push button is now the "store" button. Pressing this button stores the value on the 10 toggle switches in a FIFO.
4. When there are 5 items in the FIFO, the FIFO is drained (i.e. the 5 values are added to the previously accumulated value).
5. The 10 LEDs reflect the state of the 10 toggle switches.
6. The first FSM runs on a 5 MHz clock and manages button presses and the write side of the FIFO.
7. The second FSM runs on a 12.5 MHz clock and manages the read side of the FIFO and the accumulator.
8. This project must be implemented in VHDL and must consist of 2 FSMs, a clock-domain-crossing FIFO, and a PLL.

III. Procedure

The students began by creating the first revision of their FSM diagram, as shown in Figure 1. After working out the logic on paper, they created a copy of their Lab 5 accumulator project. Once the project was set up, the students used the Quartus Megawizard to generate the required FIFO and PLL components. With the structure of the code in place, they began coding the circuit. Throughout the semester, the students have used a modular design approach, which helped them successfully organize and implement this more complex system.

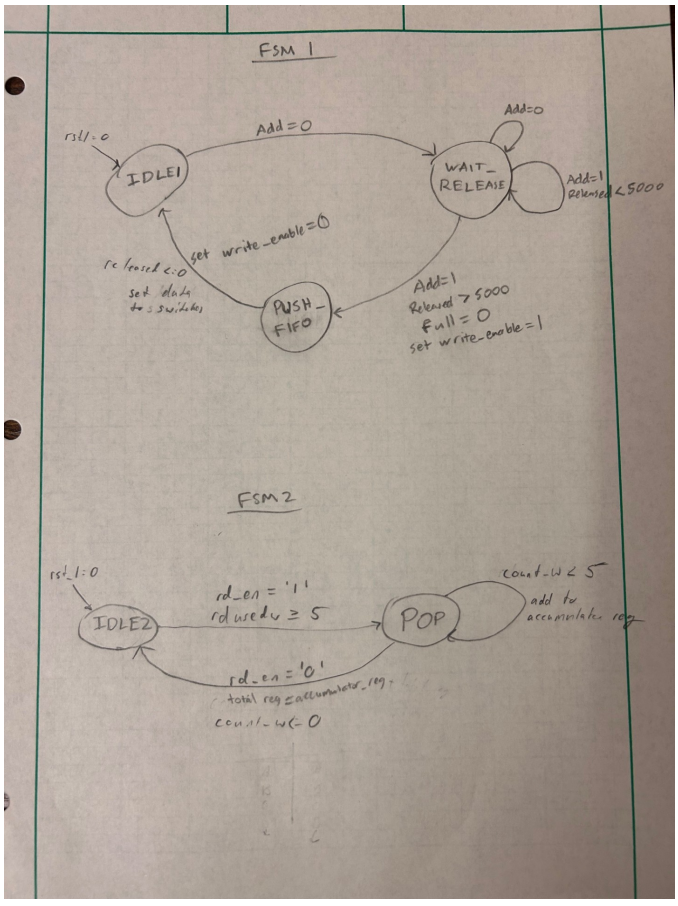

Fig. 1. Initial FSM Diagrams for the Entire System

The students then spent several hours implementing their

FSMs, troubleshooting numerous bugs along the way. While many of these issues were simple syntax errors, a few were more significant. The initial FIFO generated did not use two separate clocks, as required by the lab objectives. To correct this, the students returned to the Megawizard to reconfigure the FIFO. They later discovered another major issue—the FIFO lacked a reset signal to clear its contents. With the assistance of ChatGPT, the students learned how to implement a FIFO reset in the Top module so that the reset button would properly clear both FSMs, which included using the Megawizard again to generate a FIFO that has a clear all function built into it.

Next, the students created a quick simulation to test their circuit. The testbench revealed that the read enable signal logic was incorrect. The initial simulation results, shown in Figure 2, demonstrated that the read enable signal was cycling on and off repeatedly after five button presses. This behavior was caused by faulty logic controlling when to pull values from the FIFO.

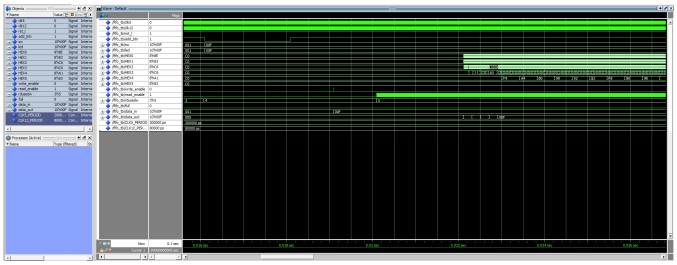

Fig. 2. Initial Simulation Showing Read Enable Logic Error

After resolving this error, the students programmed their board to test the design on hardware. Once a few minor issues were fixed, the circuit functioned as intended—the buttons, LEDs, and display all operated correctly, and the display updated after every five button presses. However, the displayed value was incorrect. The students discovered a significant bug: after a reset, the display showed the sum of only the first four button presses. After five more presses, the display showed the sum of the first nine values. The accumulator logic did not account for the clock cycle required to add each new value to the subtotal.

To correct this issue, the students returned to the drawing board and redesigned their FSMs to properly handle timing between reads and additions. The updated FSM diagrams are shown in Figures 3 and 4.

After the revised design was implimented, the system did push and pull five values from the FIFO and accumulate them properly, but on the next five button presses, a sixth value was also added, which was the last value in the previous cycle. The reasoning for this was clock delays in the logic, pulling old data before the read enable and write enables could propagate for new data. To solve this, the read-side FSM includes an additional flag that identifies when four values have been read, allowing the next read to be the final one, and ensuring the read enable signal goes low in time for the next cycle. Also, a separate flag ensures that the data is a valid input of the current

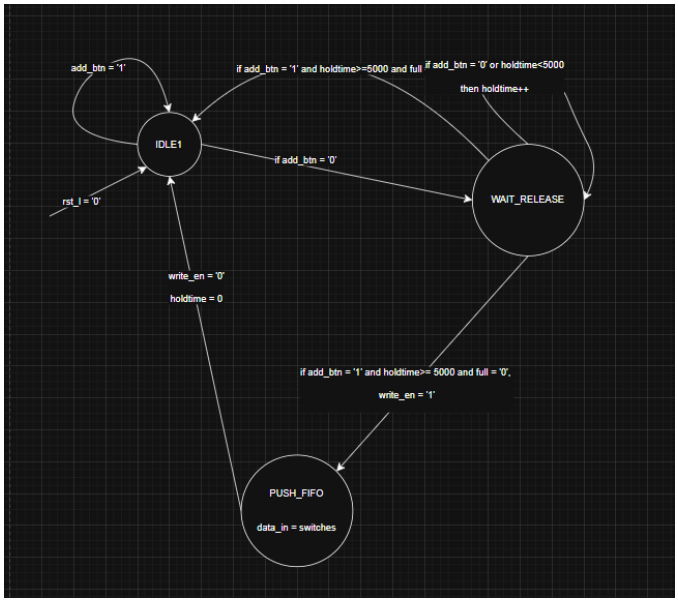

Fig. 3. FSM Diagram for the Write Side of the FIFO

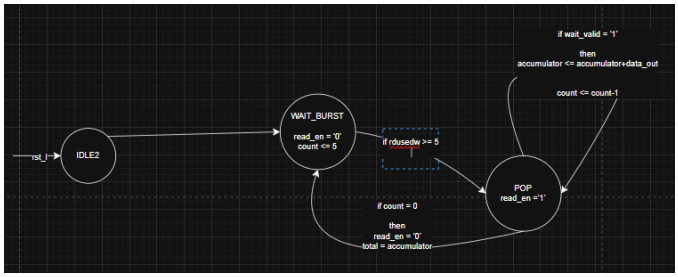

Fig. 4. FSM Diagram for the Read Side of the FIFO

5 press cycle for being used for accumulation. With these improvements implemented, the students successfully met all objectives of the lab.

IV. Results

After implementing the corrected FSM logic and reprogramming the FPGA, the system performed as expected. The FIFO correctly stored and transferred five switch values between clock domains, and the accumulator displayed the proper hexadecimal total on the 7-segment displays. The LEDs accurately mirrored the switch inputs, and the reset button successfully cleared both FSMs and the FIFO contents. The system operated reliably at both 5 MHz and 12.5 MHz, confirming that the clock-domain crossing and synchronization were implemented correctly.

V. Discussion and Conclusions

This lab demonstrated the importance of careful clock-domain management and FSM coordination in FPGA design. Early testing revealed several timing and control issues, which were resolved through improved state logic and proper FIFO configuration. The final design successfully met all objectives—implementing two FSMs, a PLL, and an asynchronous

FIFO to achieve reliable data transfer and accumulation. The project reinforced the value of modular design and simulation-driven debugging. The project also solidified the importance of planning your design before beginning to write the code. A fully-thought out design the first time would have significantly reduced the number of hours required to complete the project.

APPENDIX A
SOURCE CODE: TOP-LEVEL MODULE

This appendix includes the code for implementing the top module, including the generated PLL and FIFO.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity accumulator is
    port (
        clk5      : in std_logic;
        clk12      : in std_logic;
        rst_l     : in std_logic;
        add_btn   : in std_logic;                   -- "Add" button
        sw        : in std_logic_vector(9 downto 0); -- 10-bit input
        led       : out std_logic_vector(9 downto 0); -- mirror switches
        HEX0, HEX1, HEX2, HEX3, HEX4, HEX5 : out std_logic_vector(7 downto 0);
        write_enable : out std_logic;
        rdusedw : in std_logic_vector(2 downto 0);
        full : in std_logic;
        data_in : out unsigned(9 downto 0);
        read_enable : out std_logic;
        data_out : in unsigned(9 downto 0);
        empty : in std_logic
    );
end entity accumulator;

architecture behavioral of accumulator is
    ---------------------------------------------------------------------
    -- 7-Segment Display Lookup Table
    ---------------------------------------------------------------------
    type MY_MEM is array (0 to 15) of std_logic_vector(7 downto 0);
    constant LUT : MY_MEM := (
        X"C0", -- 0
        X"F9", -- 1
        X"A4", -- 2
        X"B0", -- 3
        X"99", -- 4
        X"92", -- 5
        X"82", -- 6
        X"F8", -- 7
        X"80", -- 8
        X"98", -- 9
        X"88", -- A
        X"83", -- B
        X"C6", -- C
        X"A1", -- D
        X"86", -- E
        X"8E"  -- F
    );

    ---------------------------------------------------------------------
    -- FSM state encodings
    ---------------------------------------------------------------------
    -- clk5 write-domain FSM
    type state_type1 is (IDLE1, WAIT_RELEASE, PUSH_FIFO);
    signal current_state1, next_state1 : state_type1;

    -- clk12 read-domain FSM
    type state_type2 is (WAIT_BURST, POP_STREAM);
    signal current_state2, next_state2 : state_type2;

    ---------------------------------------------------------------------
    -- Internal registers / signals
    ---------------------------------------------------------------------
    -- debouncer counter in clk5 domain
    signal holdtime : integer range 0 to 50000 := 0;

    -- read side bookkeeping
    signal count          : unsigned(2 downto 0) := (others => '0'); -- how many left to consume this burst
    signal accumulator_reg: unsigned(23 downto 0) := (others => '0'); -- running sum
    signal total_reg      : unsigned(23 downto 0) := (others => '0'); -- latched value shown on HEX

```

```
    -- rdreq pipeline helper
    signal read_enable_int : std_logic := '0';  -- internal rdreq
    signal wait_valid      : std_logic := '0';  -- delayed "data_out is valid now"

    -- HEX display nibbles
    signal nib0, nib1, nib2, nib3, nib4, nib5 : std_logic_vector(3 downto 0);

begin
    ------------------------------------------------------------------
    -- LED outputs mirror switches
    ------------------------------------------------------------------
    led <= sw;

    ------------------------------------------------------------------
    -- ================ clk5 DOMAIN (WRITE SIDE) ================
    -- Debounce, wait for release, push one word into FIFO
    ------------------------------------------------------------------
    process(clk5)
    begin
        if rising_edge(clk5) then
            if rst_l = '0' then
                current_state1 <= IDLE1;
                holdtime       <= 0;
                write_enable   <= '0';
                data_in        <= (others => '0');
            else
                -- advance state
                current_state1 <= next_state1;

                -- default each clk5
                write_enable <= '0';

                -- debounce timing / holdtime
                if (current_state1 = WAIT_RELEASE) and (next_state1 = WAIT_RELEASE) then
                    -- still waiting in WAIT_RELEASE
                    if (add_btn = '1') and (holdtime < 5000) then
                        holdtime <= holdtime + 1;
                    else
                        -- either button still low or we've hit 5000; holdtime stays as-is
                        holdtime <= holdtime;
                    end if;
                else
                    -- leaving WAIT_RELEASE or not in it
                    holdtime <= 0;
                end if;

                -- PUSH_FIFO: generate 1-cycle write pulse and capture switches
                if current_state1 = PUSH_FIFO then
                    write_enable <= '1';
                    data_in      <= unsigned(sw);
                end if;
            end if;
        end if;
    end process;

    -- next-state logic for clk5 FSM
    process(current_state1, add_btn, holdtime, full)
    begin
        next_state1 <= current_state1;

        case current_state1 is
            when IDLE1 =>
                if add_btn = '0' then        -- button pressed (active low)
                    next_state1 <= WAIT_RELEASE;
                else
                    next_state1 <= IDLE1;
                end if;

            when WAIT_RELEASE =>
                if (add_btn = '0') or (holdtime < 5000) then
                    -- still holding or not debounced long enough
                    next_state1 <= WAIT_RELEASE;
                elsif (add_btn = '1') and (holdtime >= 5000) and (full = '0') then
                    -- good release, fifo not full -> push
```

```
                    next_state1 <= PUSH_FIFO;
                else
                    -- either released but fifo full, or weird edge
                    next_state1 <= IDLE1;
                end if;

            when PUSH_FIFO =>
                -- 1-cycle strobe, then go chill
                next_state1 <= IDLE1;

            when others =>
                next_state1 <= IDLE1;
        end case;
    end process;


    ------------------------------------------------------------------
    -- ================ clk12 DOMAIN (READ SIDE) ================
    ------------------------------------------------------------------
    process(clk12)
        variable data_ext : unsigned(23 downto 0);
        variable depth    : unsigned(2 downto 0);
        variable will_finish_now : std_logic;
    begin
        if rising_edge(clk12) then
            if rst_l = '0' then
                current_state2   <= WAIT_BURST;
                read_enable_int  <= '0';
                wait_valid       <= '0';
                count            <= (others => '0');
                accumulator_reg  <= (others => '0');
                total_reg        <= (others => '0');
            else
                ----------------------------------------------------------
                -- advance state
                ----------------------------------------------------------
                current_state2 <= next_state2;

                ----------------------------------------------------------
                -- defaults each cycle
                ----------------------------------------------------------
                read_enable_int <= '0';

                -- sample fifo depth (safe in read clock domain)
                depth := unsigned(rdusedw);

                ----------------------------------------------------------
                -- WAIT_BURST state
                ----------------------------------------------------------
                if current_state2 = WAIT_BURST then
                    -- not actively reading
                    read_enable_int <= '0';

                    -- if we're about to leave WAIT_BURST -> POP_STREAM,
                    -- preload count with 5
                    if next_state2 = POP_STREAM then
                        count <= to_unsigned(5,3);
                        -- we are doing running accumulation, so we KEEP accumulator_reg
                        -- (if you wanted per-burst sum instead, you'd clear it here)
                    end if;
                end if;

                ----------------------------------------------------------
                -- POP_STREAM state
                ----------------------------------------------------------
                if current_state2 = POP_STREAM then
                    -- Figure out if this cycle will consume the LAST word
                    -- We "finish now" if: (a) FIFO has presented valid data this cycle
                    -- AND (b) that data is the last remaining one (count = 1 before decrement)
                    will_finish_now := '0';
                    if (wait_valid = '1') and (count = to_unsigned(1,3)) then
                        will_finish_now := '1';
                    end if;

```

```
                    -- Consume valid data_out from LAST cycle's read request
                    if wait_valid = '1' then
                        data_ext := resize(data_out, 24); -- widen 10->24
                        accumulator_reg <= accumulator_reg + data_ext;

                        -- decrement remaining count
                        count <= count - 1;
                    end if;

                    -- If we're NOT finishing this burst yet,
                    -- keep asking for more data. This is the key fix.
                    if will_finish_now = '0' then
                        read_enable_int <= '1';  -- request next word
                    else
                        read_enable_int <= '0';  -- DO NOT request another word,
                                                 -- prevents "extra" read leaking into next burst
                        total_reg <= accumulator_reg + resize(data_out,24);
                        -- note: we used accumulator_reg + data_ext above,
                        -- but data_ext is only valid when wait_valid='1'.
                        -- In the will_finish_now='1' branch, wait_valid='1', so this matches.
                    end if;
                end if;

                ------------------------------------------------------------
                -- pipeline wait_valid for next cycle
                -- wait_valid='1' means "data_out right now is from last cycle's rdreq"
                ------------------------------------------------------------
                wait_valid <= read_enable_int;
            end if;
        end if;
    end process;


    ------------------------------------------------------------------
    -- next-state logic for clk12 FSM
    ------------------------------------------------------------------
    process(current_state2, rdusedw, count, wait_valid)
        variable depth : unsigned(2 downto 0);
    begin
        depth := unsigned(rdusedw);
        next_state2 <= current_state2;

        case current_state2 is

            when WAIT_BURST =>
                if depth >= to_unsigned(5,3) then
                    next_state2 <= POP_STREAM;
                else
                    next_state2 <= WAIT_BURST;
                end if;

            when POP_STREAM =>
                -- We leave POP_STREAM after we've just consumed the last word.
                -- That's exactly when wait_valid='1' (meaning we consumed something this cycle)
                -- AND count = 1 (meaning that "something" was the last remaining one).
                if (wait_valid = '1') and (count = to_unsigned(1,3)) then
                    next_state2 <= WAIT_BURST;
                else
                    next_state2 <= POP_STREAM;
                end if;

            when others =>
                next_state2 <= WAIT_BURST;
        end case;
    end process;

    ------------------------------------------------------------------
    -- drive entity output
    ------------------------------------------------------------------
    read_enable <= read_enable_int;


    ------------------------------------------------------------------
    -- Drive HEX displays from total_reg (24 bits hex across 6 digits)
```

```
    ------------------------------------------------------------------
    nib0 <= std_logic_vector(total_reg(3  downto 0));
    nib1 <= std_logic_vector(total_reg(7  downto 4));
    nib2 <= std_logic_vector(total_reg(11 downto 8));
    nib3 <= std_logic_vector(total_reg(15 downto 12));
    nib4 <= std_logic_vector(total_reg(19 downto 16));
    nib5 <= std_logic_vector(total_reg(23 downto 20));

    HEX0 <= LUT(to_integer(unsigned(nib0)));
    HEX1 <= LUT(to_integer(unsigned(nib1)));
    HEX2 <= LUT(to_integer(unsigned(nib2)));
    HEX3 <= LUT(to_integer(unsigned(nib3)));
    HEX4 <= LUT(to_integer(unsigned(nib4)));
    HEX5 <= LUT(to_integer(unsigned(nib5)));

end architecture behavioral;
```

Listing 1. Accumulator / FIFO Controller (accumulator.vhd)

APPENDIX B
SOURCE CODE: FIFO / ACCUMULATOR FSMS

This appendix includes the accumulator code used to implement the FSMs.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity FIFO_Top is
    port (
        MAX10_CLK1_50 : in  std_logic;                        -- 50 MHz board clock
        KEY           : in  std_logic_vector(1 downto 0);     -- KEY(0)=reset, KEY(1)=add
        HEX0, HEX1, HEX2, HEX3, HEX4, HEX5 : out std_logic_vector(7 downto 0);
        LEDR          : out std_logic_vector(9 downto 0);
        SW            : in  std_logic_vector(9 downto 0)
    );
end entity FIFO_Top;


architecture top of FIFO_Top is

    ------------------------------------------------------------------
    -- Internal signals
    ------------------------------------------------------------------
    signal clk5      : std_logic;
    signal clk12     : std_logic;
    signal pll_ok    : std_logic;
    signal sys_rst_l : std_logic;  -- active-low reset after PLL lock
    signal pll_arst  : std_logic;
    signal data_in : unsigned(9 downto 0);
    signal rdreq : std_logic;
    signal wrreq : std_logic;
    signal empty : std_logic;
    signal full : std_logic;
    signal data_out : unsigned(9 downto 0);
    signal rdusedw : STD_LOGIC_VECTOR(2 downto 0);
    signal wrusedw : STD_LOGIC_VECTOR(2 downto 0);
    signal fifo_data_in_slv  : std_logic_vector(9 downto 0);
    signal fifo_data_out_slv : std_logic_vector(9 downto 0);
    signal aclr      : std_logic;  -- NEW: async clear for FIFO



    ------------------------------------------------------------------
    -- Component declarations
    ------------------------------------------------------------------
    component accumulator
        port (
            clk5    : in  std_logic;                          -- 5 MHz FSM domain
            clk12   : in  std_logic;                          -- 12.5 MHz FSM domain
            rst_l   : in  std_logic;                          -- active-low reset
            add_btn : in  std_logic;                          -- "Add" button
            sw      : in  std_logic_vector(9 downto 0);
            led     : out std_logic_vector(9 downto 0);
```

```
            HEX0, HEX1, HEX2, HEX3, HEX4, HEX5 : out std_logic_vector(7 downto 0);
            write_enable : out std_logic;
            rdusedw : in std_logic_vector(2 downto 0);
            full : in std_logic;
            data_in : out unsigned(9 downto 0);
            read_enable : out std_logic;
            data_out : in unsigned(9 downto 0);
            empty : in std_logic
        );
    end component;

    component ALT_CLKS
        port (
            areset : in  std_logic;    -- PLL async reset (active-high)
            inclk0 : in  std_logic;    -- 50 MHz input clock
            c0     : out std_logic;    -- 5 MHz clock out
            c1     : out std_logic;    -- 12.5 MHz clock out
            locked : out std_logic     -- PLL locked indicator
        );
    end component;

    component my_FIFO
        port (
            aclr        : IN STD_LOGIC  := '0';
            data        : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
            rdclk       : IN STD_LOGIC ;
            rdreq       : IN STD_LOGIC ;
            wrclk       : IN STD_LOGIC ;
            wrreq       : IN STD_LOGIC ;
            q       : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
            rdempty     : OUT STD_LOGIC ;
            rdusedw     : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
            wrfull      : OUT STD_LOGIC ;
            wrusedw     : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
        );
    end component;
begin
    pll_arst <= not KEY(0);

    fifo_data_in_slv <= std_logic_vector(data_in);
    data_out         <= unsigned(fifo_data_out_slv);

    ---------------------------------------------------------------
    -- PLL instantiation
    ---------------------------------------------------------------
    u0_pll : ALT_CLKS
        port map (
            areset => pll_arst,         -- PLL reset active-high (invert KEY(0))
            inclk0 => MAX10_CLK1_50,
            c0     => clk5,
            c1     => clk12,
            locked => pll_ok
        );

    ---------------------------------------------------------------
    -- System reset: active-low
    -- Hold logic in reset until PLL is locked and KEY(0) released.
    ---------------------------------------------------------------
    sys_rst_l <= KEY(0) and pll_ok;
    aclr      <= not sys_rst_l;      -- active-high clear to FIFO
    ---------------------------------------------------------------
    -- Accumulator / FIFO system instantiation
    ---------------------------------------------------------------
    u1_accum : accumulator
        port map (
            clk5    => clk5,
            clk12   => clk12,
            rst_l   => sys_rst_l,
            add_btn => KEY(1),
            sw      => SW,
            led     => LEDR,
            HEX0    => HEX0,
            HEX1    => HEX1,
            HEX2    => HEX2,
```

```
            HEX3    => HEX3,
            HEX4    => HEX4,
            HEX5    => HEX5,
            write_enable => wrreq,
            rdusedw => rdusedw,
            full => full,
            data_in => data_in,
            read_enable => rdreq,
            data_out => data_out,
            empty => empty
        );

        u2_my_FIFO : my_FIFO
        port map (
            data     => fifo_data_in_slv,
            rdclk    => clk12,
            rdreq    => rdreq,
            wrclk    => clk5,
            wrreq    => wrreq,
            q        => fifo_data_out_slv,
            rdempty  => empty,
            rdusedw  => rdusedw,
            wrfull   => full,
            wrusedw  => wrusedw,
            aclr => aclr
        );

end architecture top;
```

Listing 2. Top Module (fifo_top.vhd)